NAVAL POSTGRADUATE SCHOOL
Monterey, California

THESIS

TRUNCATED SUM MVL MINIMIZATION USING
THE NEIGHBORHOOD DECOUPLING ALGORITHM

by

Yao-Ming Wang

December 1989

Thesis Advisor: Chyan Yang

SECURITY CLASSIFICATION OF THIS PAGE

REPORT DOCUMENTATION PAGE

Form Approved OMB No 0704-0188

1a REPORT SECURITY CLASSIFICATION: UNCLASSIFIED	1b RESTRICTIVE MARKINGS
2a SECURITY CLASSIFICATION AUTHORITY	3 DISTRIBUTION / AVAILABILITY OF REPORT: Approved for public release; distribution is unlimited
2b DECLASSIFICATION / DOWNGRADING SCHEDULE	
4 PERFORMING ORGANIZATION REPORT NUMBER(S)	5 MONITORING ORGANIZATION REPORT NUMBER(S)
6a NAME OF PERFORMING ORGANIZATION: Naval Postgraduate School; 6b OFFICE SYMBOL (If applicable): 62	7a NAME OF MONITORING ORGANIZATION: Naval Postgraduate School
6c ADDRESS (City, State, and ZIP Code): Monterey, California 93943-5000	7b ADDRESS (City, State, and ZIP Code): Monterey, California 93943-5000
8a NAME OF FUNDING / SPONSORING ORGANIZATION; 8b OFFICE SYMBOL (If applicable)	9 PROCUREMENT INSTRUMENT IDENTIFICATION NUMBER
8c ADDRESS (City, State, and ZIP Code)	10 SOURCE OF FUNDING NUMBERS: PROGRAM ELEMENT NO; PROJECT NO; TASK NO; WORK UNIT ACCESSION NO

11 TITLE (Include Security Classification): TRUNCATED SUM MVL MINIMIZATION USING THE NEIGHBORHOOD DECOUPLING ALGORITHM

12 PERSONAL AUTHOR(S): Yao-Ming Wang

13a TYPE OF REPORT	13b TIME COVERED	14 DATE OF REPORT (Year, Month, Day)	15 PAGE COUNT
Master's Thesis	FROM _____ TO _____	1989 December	65

16 SUPPLEMENTARY NOTATION: The views expressed in this thesis are those of the author and do not reflect the official policy or position of the Department of Defense or the US Government.

17 COSATI CODES			18 SUBJECT TERMS (Continue on reverse if necessary and identify by block number)
FIELD	GROUP	SUB-GROUP	truncated sum; MVL (Multiple-Valued Logic) minimization; saturated minterm; neighborhood decoupling

19 ABSTRACT (Continue on reverse if necessary and identify by block number)

Several heuristics have been developed for the multiple-valued logic minimization problem, and while each claims some advantages in specific examples, none is significantly better than the others. Heuristic methods are interesting because exact minimization methods are extremely time-consuming. With the computer software developed at NPS called HAMLET, users can easily investigate their own heuristics.

The primary goal of this thesis is to develop an algorithm that makes the minimization of multiple-valued logic functions reasonably close to the optimal solution. The neighborhood decoupling (ND) algorithm is built on top of HAMLET. The idea of the ND algorithm is: always select the most isolated minterm as well as choose the most isolated implicant.

20 DISTRIBUTION / AVAILABILITY OF ABSTRACT	21 ABSTRACT SECURITY CLASSIFICATION	
☒ UNCLASSIFIED/UNLIMITED ☐ SAME AS RPT ☐ DTIC USERS	UNCLASSIFIED	
22a NAME OF RESPONSIBLE INDIVIDUAL: YANG, Chyan	22b TELEPHONE (Include Area Code): 408-646-2266	22c OFFICE SYMBOL: 62Ya

DD Form 1473, JUN 86 — Previous editions are obsolete. — SECURITY CLASSIFICATION OF THIS PAGE

19. continued

In this thesis, the implementation of the ND algorithm is described. A performance analysis of the ND algorithm is presented by comparing results and computation time with two published algorithms, Pomper and Armstrong's and Dueck and Miller's.

Approved for public release; distribution is unlimited

Truncated Sum MVL Minimization Using the Neighborhood Decoupling Algorithm

by

Yao-Ming Wang
Captain, Taiwan Republic of China Army
B.S., E.E., Chung Cheng Institute of Technology, 1982

Submitted in partial fulfillment of the
requirements for the degree of

MASTER OF SCIENCE IN ELECTRICAL ENGINEERING

from the

NAVAL POSTGRADUATE SCHOOL

December 1989

Author: Yao-Ming Wang

Yao-Ming Wang

Approved by: Chyan Yang

Chyan Yang, Thesis Advisor

Jon T. Butler

Jon T. Butler, Second Reader

John P. Powers

John P. Powers, Chairman
Department of Electrical and Computer Engineering

ABSTRACT

Several heuristics have been developed for the multiple-valued logic minimization problem, and while each claims some advantages in specific examples, none is significantly better than the others. Heuristic methods are interesting because exact minimization methods are extremely time-consuming. With the computer software developed at NPS called HAMLET, users can easily investigate their own heuristics.

The primary goal of this thesis is to develop an algorithm that makes the minimization of multiple-valued logic functions reasonably close to the optimal solution. The neighborhood decoupling (ND) algorithm is built on top of HAMLET. The idea of the ND algorithm is: always select the most isolated minterm as well as choose the most isolated implicant.

In this thesis, the implementation of the ND algorithm is described. A performance analysis of the ND algorithm is presented by comparing results and computation time with two published algorithms, Pomper and Armstrong's and Dueck and Miller's.

TABLE OF CONTENTS

LIST OF TABLES

LIST OF FIGURES

ACKNOWLEDGMENT

I would like to express my gratitude and appreciation to the faculty and staff of the Electrical and Computer Engineering department for providing me with the opportunity and encouragement to explore many exciting facets of electrical engineering. I would like to offer special thanks to Professor Chyan Yang for providing the necessary guidance and direction in the formulation of this document. This work has benefited immensely from the work of HAMLET by Yurchak and Butler. I also wish to thank Professor Jon T. Butler for his valued assistance as my second reader.

Finally, I am most grateful to my wife Pei-Chun for her patience, understanding, and support during my studies .

I. INTRODUCTION

A. MOTIVATION

Binary computing systems have been used for about 30 years. The very-large-scale-integration (VLSI) technology has resulted in increasingly smaller circuits. However, the pace of reduction has showed because of two major problems, bus connection and pin limitation. In recent years, multiple-valued logic has been proposed to solve those problems [Ref. 1, 2]. Since the programmable logic array (PLA) is a basic tool for binary VLSI design, it is expected to be important in multiple-valued logic (MVL). This observation has inspired the work reported in this thesis.

The truncated sum (TSUM) operation of multiple-valued logic is easily implemented in the charge-coupled devices (CCD) technology [Ref. 3, 4]. Several heuristics have been developed for the multiple-valued logic minimization involving TSUM and each claims some advantages in specific examples, but none of them is consistently better than the others [Ref. 5, 6, 7, 8, 9, 10]. Heuristic methods are interesting because of exact minimization methods are extremely time-consuming. Until now, only Tirumalai and Butler [Ref. 9] have analyzed the performance of different MVL minimization algorithms. Their study is limited to two-variable four-valued functions. Three open questions exist:

1. In the case of two-variable four-valued functions, can we design an algorithm which is better than previously proposed algorithms?
2. What is the algorithm performance if we increase the number of variables (greater than two)?
3. What are the computation time requirements for the various algorithms?

With the computer software developed at Naval Postgraduate School called HAMLET (Heuristic Analyzer for Multiple-valued Logic Expression Translation) [Ref. 10],

users can easily investigate their own heuristics. For example, the neighborhood decoupling (ND) algorithm described in this thesis is built as one independent option of HAMLET. This thesis reexamines the algorithms used in HAMLET and analyzes the new method, the ND algorithm. It also investigates each algorithm under four different function settings and summarizes their performance and computation time.

B. BACKGROUND

The heuristic approaches in HAMLET can all be classified as greedy algorithms. The HAMLET execution procedure of these algorithms is abstracted as follows. Formal definitions will be covered in the next chapter. Let f be a multiple-valued function and let α, a minterm, be an assignment of values to the variables of f such that $f \neq 0$.

```
/* ************************************************************************
     Input: let the M be the set of minterms of a function f;
     Output: the minimized sum of product, S, of the original function:
************************************************************************ */
{
     S ← φ.
      While (M ≠ φ) do {
            pick one minterm α from M;
            find an implicant I_α which covers α;
            S ← I_α ∪ S;
            subtract I_α from f;
           }
}
```

TABLE 1.1: Summary of Three Heuristic Algorithms

Heuristic Algorithm	Choice of Minterm	Choice of Implicant
Pomper and Armstrong [Ref. 5] (1981)	Random	Drives Most Minterms to 0 or *don't-care*
Besslich [Ref. 6] (1986)	Smallest Weight (Most Isolated)	Drives Most Minterms to 0 or *don't-care*
Dueck and Miller [Ref. 7, 8] (1988)	Largest IF (Most Isolated)	Largest BCR

Table 1.1 shows three previously proposed algorithms. Each algorithm differs from the others in the manner of picking the minterms (α) and finding the implicants (I_α). For example, in the above recursive search procedure, the input function expression f is evaluated at minterm α. Next an implicant I_α is chosen which covers α. Then, implicant I_α is added to output solution set S, and is subtracted from function f.

Pomper and Armstrong introduced a heuristic algorithm that picks α randomly (as long as α is in the set of minterm M) and finds I_α (as long as I_α covers α) which drives the most minterms to 0 or *don't-care* when I_α subtracted from function f [Ref. 5]. In 1986, Besslich presented a direct cover algorithm, according to weight transformations. The Besslich algorithm picks α with the smallest weight (most isolated minterm) and finds I_α which has a lowest cost per minterm covered (i.e., which drives the most minterms to 0 or *don't-care*) [Ref. 6]. In 1988, Dueck and Miller presented another algorithm that picks α from M if α has the highest isolated factor (IF) and then finds the I_α which directly covers α such that the break count reduction (BCR) is maximum [Ref. 7, 8].

Each algorithm has some advantages in specific examples [Ref. 9]. The ND algorithm is characterized by adopting the advantage from each algorithm and fully utilizing the properties of the truncated sum. The ND algorithm is an improvement to the Dueck and Miller algorithm with revised decision rules for making selections of minterms and implicants.

C. THESIS OUTLINE

A summary of MVL definitions for truncated sum minimization are introduced in Chapter II. The notations and definitions of Chapter II also help us in explaining the algorithms appearing in subsequent chapters. The neighborhood decoupling algorithm is presented in Chapter III. Chapter IV and V discuss the performance and computation times of the neighborhood decoupling algorithm with Pomper and Armstrong's and Dueck and Miller's algorithm for different function parameters.

II. NOTATIONS AND DEFINITIONS

Although multiple-valued logic minimization for truncated sum has been discussed in resent years [Ref. 7, 8, 9, 10, 11], a distinct and complete formal definition for truncated sum MVL minimization is not available. In view of this, we studied the truncated sum MVL minimization, then summarized and give several definitions that will be presented in Section A. It will also help us to explain the algorithm given in the next chapter. We will report our observations on truncated sum properties in Section B, and the definitions for ND algorithm will be discussed in Section C.

A. DEFINITIONS FOR TRUNCATED SUM

Definition 1:

Let $\mathcal{X} = \{ x_1, x_2, \ldots, x_n \}$ be a set of n input variables where x_i takes on values from $\mathcal{R} = \{0, 1, \ldots, r-1\}$. An n-variable r-valued function f is a mapping

$$f : \mathcal{R}^n \longrightarrow \mathcal{R} \cup \{r\}. \; [Ref. 9]$$

Here, r is a $don't - care$ value; it can be chosen freely from any of the logic values, $0, 1, \ldots, r-1$.

Definition 2: MIN

The MIN [Ref. 9] function is denoted as $f(x_1, x_2) = x_1 x_2$ which evaluates to the minimum value of its arguments. For example, if $\mathcal{R} = \{0,1,2,3\}$, then $f(1,2) = 1$ and $f(0,3) = 0$. A minterm is an assignment of values to $x_1, x_2, \ldots, x_n$ such that $f(x) \neq 0$.

Definiton 3: Literal

The literal operation of a variable x is defined as:

$$ {}^{a}x^{b} = \begin{cases} r-1 & \text{if } a \leq x \leq b \\ 0 & \text{otherwise. [Ref. 11]} \end{cases} \tag{2.1} $$

Definition 4: Truncated Sum (TSUM)

The truncated sum (TSUM) operation is defined as:

$$ TSUM(x_1, x_2) = x_1 + x_2 = min(x_1 + x_2, r-1). \; [Ref.\ 11] \tag{2.2} $$

The two + signs in this expression are different. The leftmost denotes the TSUM operation, while the rightmost denotes ordinary add two of logic values which are viewed as integer. The TSUM obeys the associative and commutative rules. For example, if $\mathcal{R}$ = {0,1,2,3}, then TSUM(1,2)=3 and TSUM(2,2)=3.

These definitions are inspired by the fact that CCD implementation supports TSUM naturally [Ref. 3, 4].

Example 1:

For example, ${}^{1}x_1^3$ is a literal and takes value of 3 when $1 \leq x_1 \leq 3$. However, function $2\ {}^{1}x_1^3$ takes a value of 2 based on to the definition of MIN. Similarly, the product term $2\ {}^{3}x_1^3\ {}^{0}x_2^0$ is a function that is 2, when $x_1 = 3$ and $x_2 = 0$.

Definition 5: Product Term

A product term p is the MIN of one nonzero constant $c \in \mathcal{R}$, and one or more literal functions. In general, a product term is defined as:

$$ p \equiv c\ {}^{i_1}x_1^{j_1}\ {}^{i_2}x_2^{j_2} \ldots {}^{i_n}x_n^{j_n} \begin{cases} i_k \leq j_k \\ i_k, j_k \in \mathcal{R}; \quad 1 \leq k \leq n. \end{cases} \tag{2.3} $$

The constant or coefficient c, in a product term, effectively scales the term. For each variable x_i, we say the window size of the literal ${}^{i_k}x_i^{j_k}$ is $j_k - i_k + 1$. We use the terms *product term* and *implicant* interchangeably in this thesis.

Definiton 6: Sum-of-Products Expression

A sum-of-products expression is $p_1 + p_2 + \ldots + p_N$ for some integer N, where p_i is a product term.

Definition 7: Minterm

A minterm α is of the form $c\ ^{a_1}x_1^{a_1}\ ^{a_2}x_2^{a_2}\ ^{a_3}x_3^{a_3}\ \ldots\ ^{a_n}x_n^{a_n}$ where $a_i \in \mathcal{R}$ and constant $c \in \mathcal{R} - \{\ 0\ \}$. We say the coordinate of α is $< a_1, a_2, \ldots a_n >$. We denote the value of minterm α, $g(\alpha)$, as the nonzero constant c.

A product term $p = c\ ^{i_1}x_1^{j_1}\ ^{i_2}x_2^{j_2} \ldots\ ^{i_n}x_n^{j_n}$ can be decomposed into $\prod_{k=1}^{n}(j_k - i_k + 1)$ minterms. We say p generates those minterms. Given a product term p, the set of minterms generated from p is denoted by MS_p. If the number of elements in MS_{p_1} is greater than that in MS_{p_2}, we say p_1 covers a larger area than p_2. Given a function f, the set of minterms generated from its product terms is denoted by MS_f.

Definition 8: Saturated Minterms (SAT)

Given a minterm α generated from the original function to be minimized, if $g(\alpha) = r - 1$, then α is a saturated minterm. Let SAT be the set of all saturated minterms of a function.

Example 2:

If the input function to be minimized is expressed as follows,

$$f = 3\ ^{0}x_1^{3}\ ^{1}x_2^{1} + 2\ ^{1}x_1^{2}\ ^{0}x_2^{0} + 3\ ^{1}x_1^{1}\ ^{2}x_2^{3} + 2\ ^{2}x_1^{2}\ ^{2}x_2^{3} + 1\ ^{2}x_1^{2}\ ^{3}x_2^{3} + 1\ ^{0}x_1^{0}\ ^{2}x_2^{2},$$

the MS_f can be represented as 11 minterms in Figure 2.1. We mark a saturated minterm with a dot in the figure.

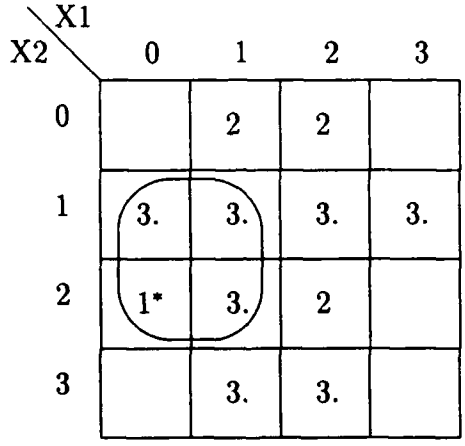

Figure 2.1: Map for Example 2, 3, 4; Step 1 of Table 3.2

Lemma 1 *Given a minterm α the maximum number of implicants which covers α is $O(r^{2n})$.*

Proof: Consider a variable (axis) x_i of α. Any implicant (I_α) that covers α may have a range or "window size" w, such that $1 \leq w \leq r$. With a window size w, we may have w implicants that covers α. That is, the minterm α can be at any position of the window. For a given axis, counting all window sizes, we have $1 + 2 + 3 + \ldots + r = \frac{r(r+1)}{2}$ possible implicants that cover α. Over the entire n-dimensional space, we have $(\frac{r(r+1)}{2})^n = O(r^{2n})$ [Ref. 12].

B. THE PROPERTIES OF TRUNCATED SUM

There are two important properties of the truncated sum which are useful later in developing the ND algorithm.

1. Saturated minterm can be generated by TSUM operation.

The truncated sum of two or more minterms may produce a saturated minterm. By definition 4, the truncated sum of any minterm and a saturated minterm remains a saturated minterm. In other words, given two minterms α, β and minterm $\gamma = TSUM(\alpha, \beta)$. If value of γ is $r - 1$, i.e., γ is a saturated minterm then for any other minterm δ, $\gamma + \delta = \gamma$.

As an example, in a 2-variable 4-valued function, three minterms add in one position.

$$2\ {}^1x_1^1\,{}^2x_2^2 + 2\ {}^1x_1^1\,{}^2x_2^2 + 1\ {}^1x_1^1\,{}^2x_2^2 = 3\ {}^1x_1^1\,{}^2x_2^2 + 1\ {}^1x_1^1\,{}^2x_2^2 = 3\ {}^1x_1^1\,{}^2x_2^2$$

The first two terms form a saturated minterm, and this saturated minterm absorbs the third minterm.

2. *Don't care* **minterm can be produced by saturated minterm.**

   In the minimization procedure, we may update a minterm α to α' by subtracting minterm γ ($\alpha' = \alpha - \gamma$), where γ is the value of selected implicant. If $\alpha \in SAT$, in succession of updates, the value of α' may reach the value 0. In that case, the algorithm will reset that minterm coordinate to *don't-care*, i.e., value r.

C. DEFINITIONS USED IN ND ALGORITHM

Definition 9: Direct Neighbors

Let α and β be minterms with coordinates $< a_1, a_2, \ldots a_n >$ and $< b_1, b_2, \ldots b_n >$ respectively. If for all i we have $a_i = b_i$ except one position j such that $|a_j - b_j| = 1$ we say that α and β are direct neighbors. Given a minterm α, we use $N(\alpha)$ to denote the set of its direct neighbors.

Observation 1: *The maximum number of direct neighbors of a given minterm is* $2n$.

Definiton 10: Directional Neighbors

Two minterms α and β are directional neighbors in the direction x_j, if $a_i = b_i$ for all $i \in [1, n]$ such that $i \neq j$. When $b_j > a_j$ we say that β is in the positive direction of α, while when $b_j < a_j$ we say that β is in the negative direction of α.

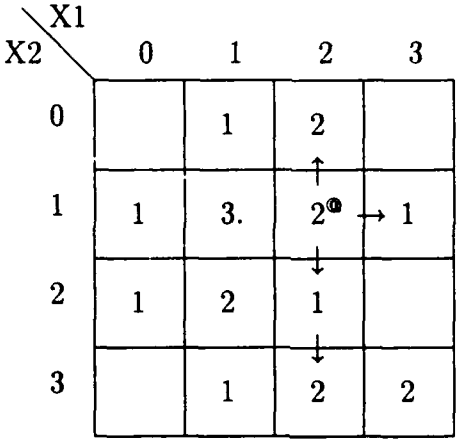

Figure 2.2: Example for Connected Minterms

Observation 2: *If β is a direct neighbor of α then β is a directional neighbor of α in the direction of x_i for some $i \in [1, n]$.*

Definition 11: Connected Minterms

This is a recursive definition. Given a minterm α and a minterm β, then we say β *is a connected minterm of* α, if

1. β is a direct neighbor of α and either $g(\beta) \leq g(\alpha)$ or $\alpha \in SAT$.
2. β is a directional neighbor of α in direction x_i and β's direct neighbor is connected to α *and* either $g(\beta) \leq g(\alpha)$ or $\alpha \in SAT$.

For example, in Figure 2.2 minterm $2\ {}^2x_1^2\,{}^0x_2^0$, $1\ {}^3x_1^3\,{}^1x_2^1$, $1\ {}^2x_1^2\,{}^2x_2^2$ and $2\ {}^2x_1^2\,{}^3x_2^3$ (pointed by arrow) are connected minterms of $2\ {}^2x_1^2\,{}^1x_2^1$ (the minterm with @ sign).

Definition 12: Connected Minterm Count

CMC_α is the connected minterm count of minterm α. It is the number of minterms that are connected to minterm α.

Definition 13: Expandable Directional Count

EDC_α is the expandable directional count of minterm α. It is the number of directions (both positive and negative for each x_i) in which α has one or more connected minterms.

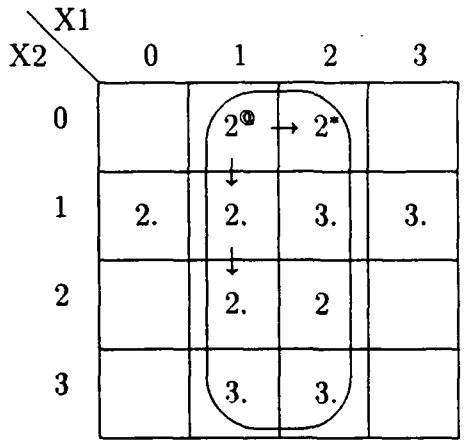

Figure 2.3: Map for Example 3, 4; Step 2 of Table 3.2

Observation 3: $0 \leq EDC_\alpha \leq 2n$.

Definition 14: Clustering Factor

The clustering factor relative to a minterm α is defined as

$$CF_\alpha = EDC_\alpha(r-1) + CMC_\alpha. \tag{2.4}$$

This is a measure of the weight of all connected minterms relative to α. The $(r-1)$ factor is the range, or maximum possible number of minterms, in a direction x_i.

Example 3:

In Figure 2.1 the minterm $1\ {}^0x_1^0\,{}^2x_2^2$ (the minterm with * sign) has no connected minterms nor expandable directional neighbors, i.e., its CMC and EDC values are 0. Figure 2.3 shows that a circled implicant $1\ {}^0x_1^1\,{}^1x_2^2$ was subtracted from Figure 2.1. We mark a minterm with a dot in figure because it was a saturated minterm in original function map (see Definition 8 and Figure 2.1). The minterm $\alpha = 2\,{}^1x_1^1\,{}^0x_2^0$ (the minterm with @ sign) is one of the ten minterms and $CMC_\alpha = 3$, $EDC_\alpha = 2$. The clustering factors of all minterms in Figure 2.3 are listed in Table 2.1.

TABLE 2.1: *CFs* **for all minterms in Figure 2.3**

Minterm	$2^{1}x_1^{1}0x_2^{0}$	$2^{2}x_1^{2}0x_2^{0}$	$2^{0}x_1^{0}1x_2^{1}$	$2^{1}x_1^{1}1x_2^{1}$	$3^{2}x_1^{2}1x_2^{1}$
CF	9	4	6	18	18
Minterm	$3^{3}x_1^{3}1x_2^{1}$	$2^{1}x_1^{1}2x_2^{2}$	$2^{2}x_1^{2}2x_2^{2}$	$3^{1}x_1^{1}3x_2^{3}$	$3^{2}x_1^{2}3x_2^{3}$
CF	6	13	4	10	10

III. NEIGHBORHOOD DECOUPLING ALGORITHM

The neighborhood decoupling (ND) algorithm [see Appendix A for the C program listing] is an improvement to Dueck and Miller's method [Ref. 8]. Like the algorithms mentioned in Chapter I Section B and throughout, the ND algorithm has two computational phases: select a minterm and select an implicant. Firstly, the most isolated minterm is chosen by using the algorithm M to be described in Section A. The most isolated minterms in general are different from Dueck and Miller's method due to different decision rules. Secondly, from all implicants which cover the most isolated minterm ND algorithm chooses the one that is not strongly "coupled" with its neighbors. In other words, it is the most isolated implicant. This decoupling process is based on the observation that if we choose that specific implicant then we may minimize the negative impact for future minterm selections as well as implicant selections. The explanation of this idea is described in Section B. In the algorithm below, f denotes the function to be minimized.

{

$SS \leftarrow \phi$; /* SS = Solution Set */

$WS = MS_f = \{\alpha | \alpha$ is generated by the function f; if $\alpha \in SAT$ then mark its coordinate $\}$.

While $WS \neq \phi$ **do** {

1. Use algorithm M (see Section A) to select a minterm α from the WS.
2. Use algorithm N (see Section B) to select an implicant I_α that covers α.
3. $SS \leftarrow SS \cup I_\alpha$.

4. $\forall \beta \in I_\alpha$ **do** {

compute $g(\beta) \leftarrow g(\beta) - g(\alpha)$.

subtracted I_α from WS.

if β is originally marked and $g(\beta) = 0$ **then** $g(\beta) \leftarrow r$.

/* don't-care terms */

}

}

}

A. ALGORITHM M: MINTERM SELECTION

Based on Definition 15, all minterm's clustering factor can be computed. The ND algorithm computes the clustering factor in the order of coordinates (x_i), i.e., row-column order. For example, the minterm $2\ ^2x_1^2\ ^0x_2^0$ is evaluated earlier than the minterm $2\ ^2x_1^2\ ^2x_2^2$. The algorithm M is described as follows:

1. Compute the corresponding CF_{α_k} for all $\alpha_k \in WS$.
2. Select the minterm α that has the smallest clustering factor (excluding the minterm $\alpha \in SAT$ as well as greater than $2 \times (r-1)$). If there is a tie, the *first* one that gets evaluated is chosen.

B. ALGORITHM N: NEIGHBORHOOD RELATIVE COUNT

The purpose of algorithm N is to choose the most "isolated" implicant (I_α) and update the working set WS. It computes the neighborhood relative count (NRC) for all implicants that cover the minterm α. The implicant with the smallest NRC is chosen. In other words, NRC is a measure of the coupling strength of an implicant with its neighbors. To select an implicant which is equivalent to breaking the coupling between that implicant with its neighbors, the candidate implicant should have the smallest coupling strength with its neighbors. Therefore, the ND algorithm tends to choose the most "isolated" implicant. If there is a tie in selecting the I_α, ND algorithm chooses the one which

covers the *largest area.* The computation of NRC for a given implicant is described as follows:

1. Initialize the NRC to zero.
2. Check all neighboring minterms of the implicant and increment or decrement its NRC according to the following (intuitively stated) rule, which is, if the coupling strength between covered and uncovered area is weak (good for further decoupling), algorithm N decreases NRC, otherwise increases NRC.

/* **

α: the chosen minterm from algorithm M.

M: the set of minterms which was covered (generated) by the chosen implicant (I_α).

$N(\beta)$: the set of direct neighbors of minterm β.

** */

{

$NRC \leftarrow 0$;

$\forall \beta \in M$ **and** $\beta \neq \alpha$ **do** {

if$(g(\beta) - g(\alpha) \leq 0)$ **then** $NRC \leftarrow NRC - 2$;

}

$\forall \beta \in M$ **and** $\forall \gamma \in N(\beta)$ **do** {

if ($\gamma \notin M$ **and** $\gamma \neq 0$ **and** ($\gamma \notin SAT$ **or** $\beta \notin SAT$)) **then** {

if ($g(\beta) - g(\alpha) > g(\gamma)$) **then** {

if ($\gamma \in SAT$) **then** $NRC \leftarrow NRC - 1$;

else $NRC \leftarrow NRC + 2$;

}

if ($g(\beta) - g(\alpha) < g(\gamma)$) **then** {

if ($g(\beta) = g(\gamma)$) **then** $NRC \leftarrow NRC + 2$;

```
                    if (γ ∈ SAT and g(γ) − g(β) < 0 ) then
                                NRC ← NRC + 2;
                        else {
                        if (g(β) > g(α) and g(β) ≠ g(γ) ) then {
                          if( β ∈ SAT) then  NRC ← NRC − 1;
                                        else NRC ← NRC + 2;
                          } /* end if */
                      } /* end else */
                  } /* end if */

              if (g(β) − g(α) = g(γ) ) then {
                    if (g(γ) > 0 or β ∈ SAT) then
                                NRC ← NRC − 1;
                          else NRC ← NRC − 2;
                  }
            } /*end if */
        }
    if (M = {α}) then {
        if (α ∈ SAT) then NRC ← 2;
            else if (NRC < 0) then NRC ← 1;
     }
     else NRC ← NRC + 2;
}
```

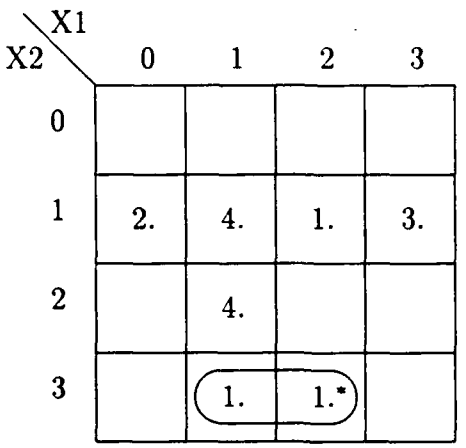

Figure 3.1: Map for Example 4; Step 3 of Table 3.2

Example 4:

It is instructive to examine an example of the application of the ND algorithm. We use a two-variable four-valued function as an example. The input function to be minimized is expressed as:

$$f = 3\ {}^{0}x_1^{3}\,{}^{1}x_2^{1} + 2\ {}^{1}x_1^{2}\,{}^{0}x_2^{0} + 3\ {}^{1}x_1^{1}\,{}^{2}x_2^{3} + 2\ {}^{2}x_1^{2}\,{}^{2}x_2^{3} + 1\ {}^{2}x_1^{2}\,{}^{3}x_2^{3} + 1\ {}^{0}x_1^{0}\,{}^{2}x_2^{2}.$$

The working set, WS, is initialized to MS_f and is represented in Figure 2.1. The clustering factors of all minterms in WS are calculated (see example 3 for computation). The smallest CF that first comes from minterm $1\ {}^{0}x_1^{0}\,{}^{2}x_2^{2}$; therefore, algorithm M will select $\alpha = 1\ {}^{0}x_1^{0}\,{}^{2}x_2^{2}$. The ND algorithm computes the NRC_I for each implicant I which covers α using algorithm N. Since implicant $1\ {}^{0}x_1^{1}\,{}^{1}x_2^{2}$ has the smallest $NRC(-2)$, the ND algorithm selects it as the first implicant in the solution set (SS). Table 3.2, together with Figures 2.3, 3.1, and 3.2 shows the steps of choosing successive implicants. The * sign in each figure indicates the most isolated minterm while a circled implicant is the most isolated implicant. Suppose we have chosen two implicants from the function shown in Figure 2.1 the resulting function is shown in Figure 3.1. The minterm $1\ {}^{2}x_1^{2}\,{}^{3}x_2^{3}$ is selected, since it has the smallest $CF(4)$.

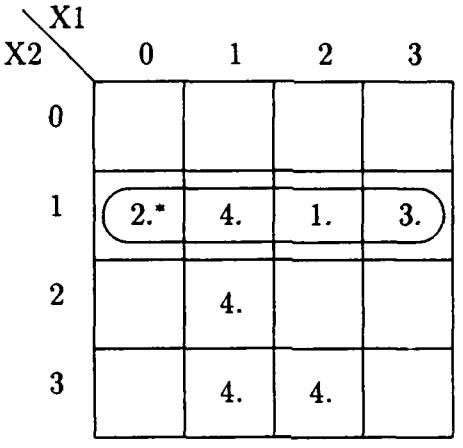

Figure 3.2: Map for Example 4; Step 4 of Table 3.2

TABLE 3.1: *NRCs* **for Minterm** $2\,^{0}x_1^{0}\,^{1}x_2^{1}$ **in Figure 3.2**

Implicant	$2\,^{0}x_1\,^{0}\,^{1}x_2\,^{1}$	$2\,^{0}x_1\,^{1}\,^{1}x_2\,^{1}$	$2\,^{0}x_1\,^{2}\,^{1}x_2\,^{1}$	$3\,^{0}x_1\,^{3}\,^{1}x_2\,^{1}$
NRC	2	0	-2	-4

There are two implicants that cover the minterm $1\,^{2}x_1^{2}\,^{3}x_2^{3}$, and their NRC values are 2 and -2. The implicant $1\,^{1}x_1^{2}\,^{3}x_2^{3}$ is chosen since it has the smallest $NRC(-2)$. Having updated the working set and added $1\,^{1}x_1^{2}\,^{3}x_2^{3}$ to the solution set, we have the new map in Figure 3.2. From Figure 3.2, the NRC values for the minterm $2\,^{0}x_1^{0}\,^{1}x_2^{1}$ are available in Table 3.1. Finally, the working set should contain value 0 (empty square) or 4 (*don't-care*) as shown in Figure 3.3.

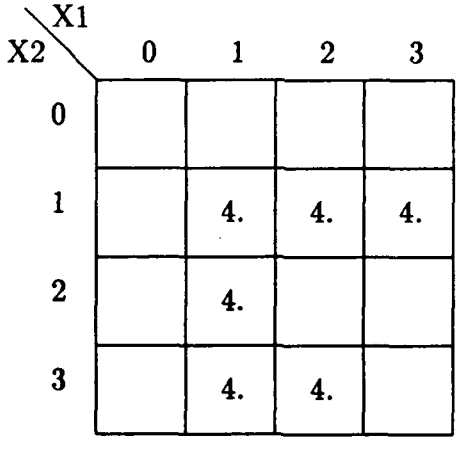

Figure 3.3: Final Working Set

TABLE 3.2: Steps of ND algorithm

Step	Minimum CF_α	Minterm α	Minimum NRC	Candidate Implicant	Reference
1	0	$2\ ^{0}x_1{}^{0}\ ^{2}x_2{}^{2}$	-2	$2\ ^{0}x_1{}^{1}\ ^{1}x_2{}^{2}$	Figure 2.1
2	4	$1\ ^{2}x_1{}^{2}\ ^{0}x_2{}^{0}$	-8	$1\ ^{1}x_1{}^{2}\ ^{0}x_2{}^{3}$	Figure 2.3
3	4	$1\ ^{2}x_1{}^{2}\ ^{3}x_2{}^{3}$	-2	$1\ ^{1}x_1{}^{2}\ ^{3}x_2{}^{3}$	Figure 3.1
4	6	$2\ ^{0}x_1{}^{0}\ ^{1}x_2{}^{1}$	-4	$3\ ^{0}x_1{}^{3}\ ^{1}x_2{}^{1}$	Figure 3.2

The final minimized result, g, is expressed as:

$$g = 2\ ^{0}x_1^{1}\ ^{1}x_2^{2} + 1\ ^{1}x_1^{2}\ ^{0}x_2^{3} + 1\ ^{1}x_1^{2}\ ^{3}x_2^{3} + 3\ ^{0}x_1^{3}\ ^{1}x_2^{1}.$$

IV. COMPARISON RESULTS

In this thesis all testing results were obtained by running sample functions on the VAX 11/785 and ISI workstations. A large number of sample functions were randomly generated (63,500). We applied each algorithm to minimize these sample functions in a way similar to the method used by Tirumalai and Butler [Ref. 9], and Yurchak and Butler [Ref. 10]. However, Tirumalai and Butler generated the input functions with a fixed number of minterms. We generated the input function with a fixed number of product terms. This thesis investigated three algorithms: (1) Pomper and Armstrong [Ref. 5], (2) Dueck and Miller [Ref. 7, 8, 10], and (3) Neighborhood Decoupling under various settings:

1. Two-variable four-valued with 3 to 16 input product terms.
2. Two-variable five-valued with 3 to 25 input product terms.
3. Three-variable four-valued with 3 to 45 input product terms.
4. Four-variable four-valued with 3 to 35 input product terms.

For two-variable four-valued functions, we generated 1000 sample functions for a given number of input product terms from 3 to 16, i.e., we generated 14,000 functions to test the different algorithms. For the other three, we generated 500 functions for each given number of input product terms, i.e., for each set we generated 11,500, 21,500, and 16,500 functions respectively.

In Section B, functions with a different number of product terms are tested and the computation times are recorded. For a given number of product terms, the computation time from two to four variable functions grows exponentally. This explains why we did not produce more than 1000 functions for each case.

A. PERFORMANCE COMPARISON

There are two ways to measure the performance. First, the average number of output product terms will show us the advantage of each algorithm. Second, the ratio of the performance results will show us the performance increases or decreases for each algorithm under different conditions.

1. AVERAGE NUMBER OF OUTPUT PRODUCT TERMS

For each set, we compute the average number of output product terms (see Appendix B). From these data points, we can plot a curve, indicating the average number of output product terms as a function of input product terms. The plots for the various sets are shown in Figures 4.1, 4.2, 4.3, and 4.4. In these bell-shaped figures we observed three important features:

1. The differences in the number of average output product terms among those algorithms are insignificant, when the input number of product terms are small. When the number of input product terms is small, the function is simple and easy to minimized; therefore, the number of output product terms for each algorithm are almost same.
2. The highest point of each bell-shaped curve is the highest average number of output product terms for each algorithm. The number of input product terms that has the highest average number of output product terms constitutes the most complicated functions.
3. When the number of input product terms get larger, the curve goes down gradually. This is chiefly due to the fact that the more input product terms in a function the higher tendency of generating saturated minterms. In most cases, a single implicant can cover a cluster of saturated minterms.

From Figures 4.1, 4.2, 4.3, and 4.4, the neighborhood decoupling algorithm obviously outperforms the other two algorithms.

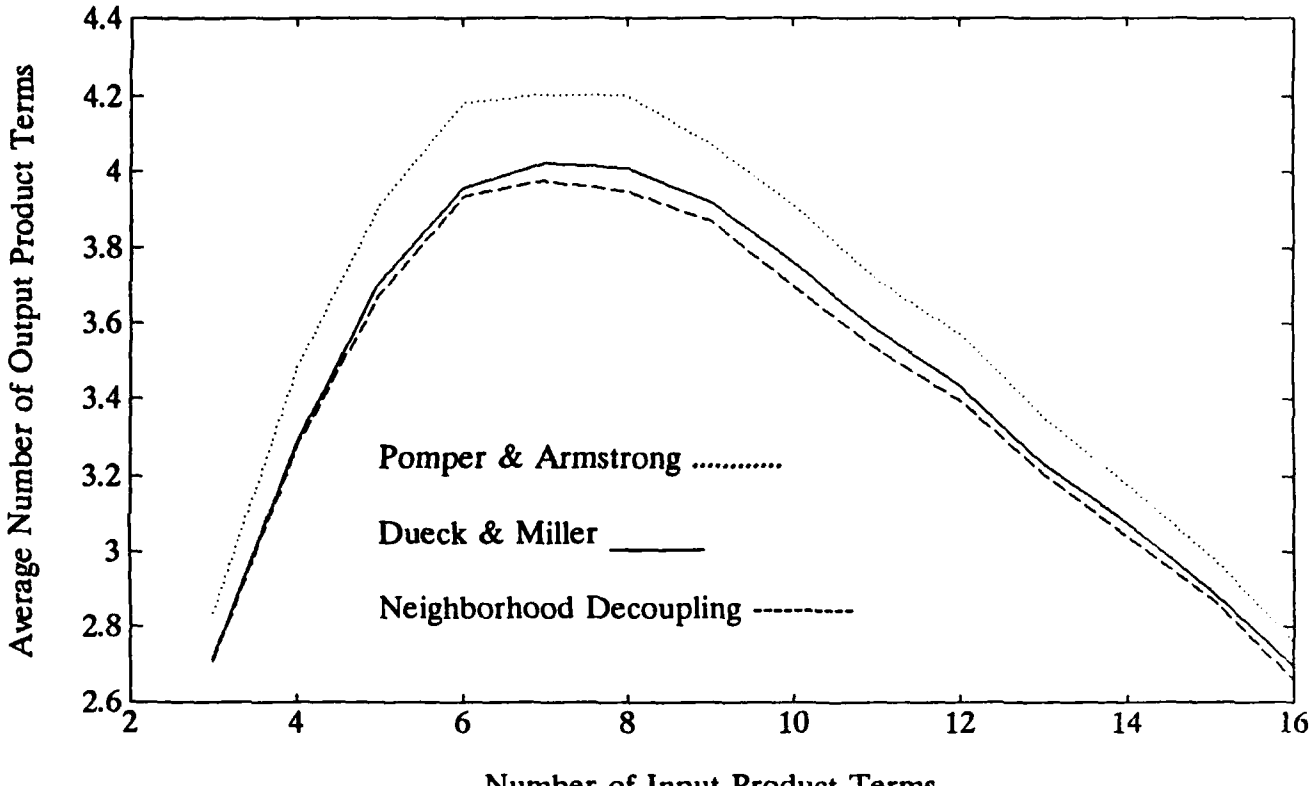

Figure 4.1: Two-Variable Four-Valued Average Product Term

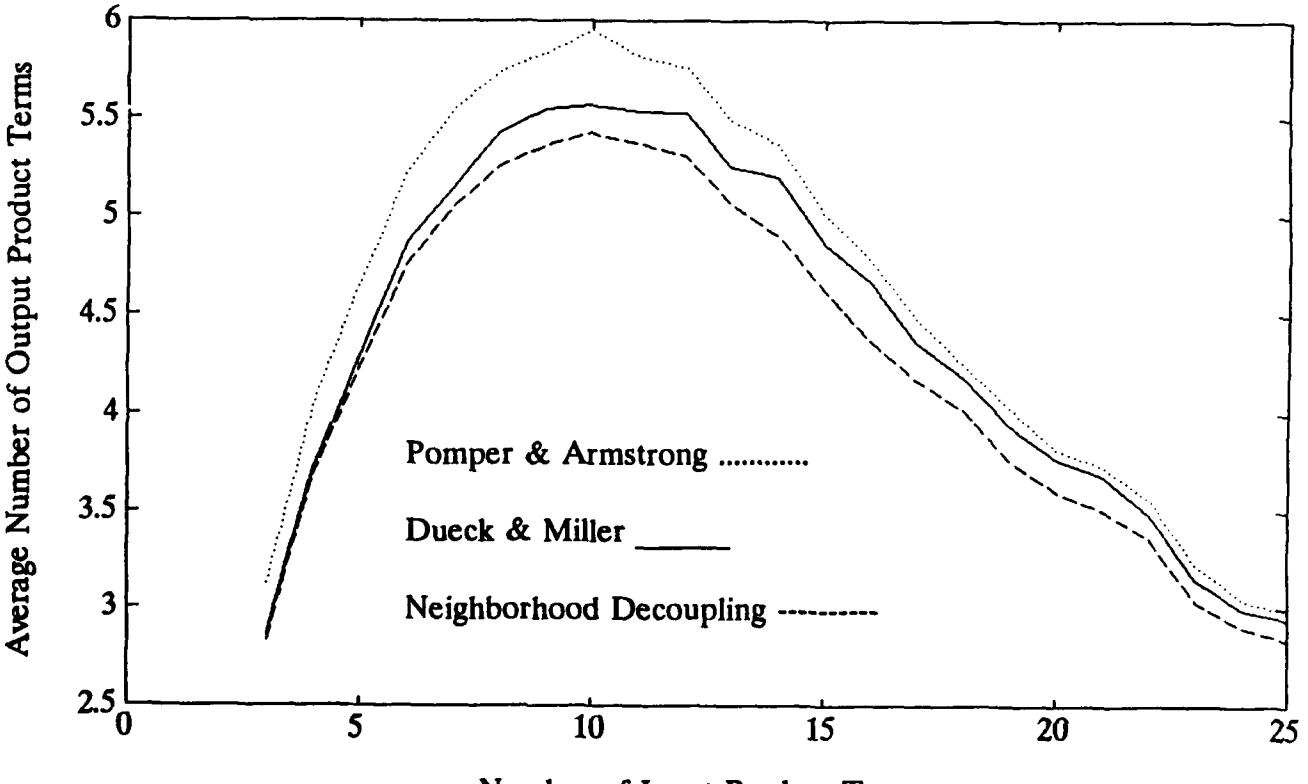

Figure 4.2: Two-Variable Five-Valued Average Product Term

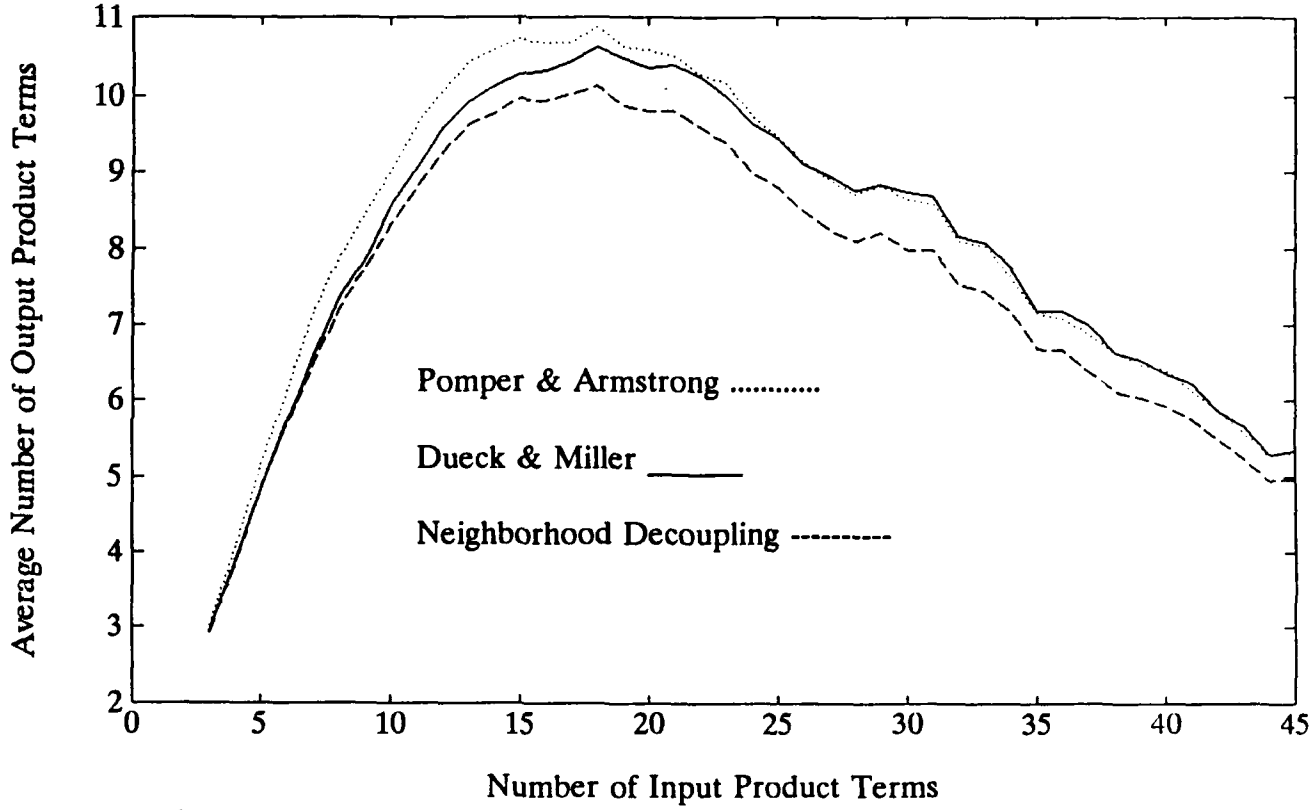

Figure 4.3: Three-Variable Four-Valued Average Product Term

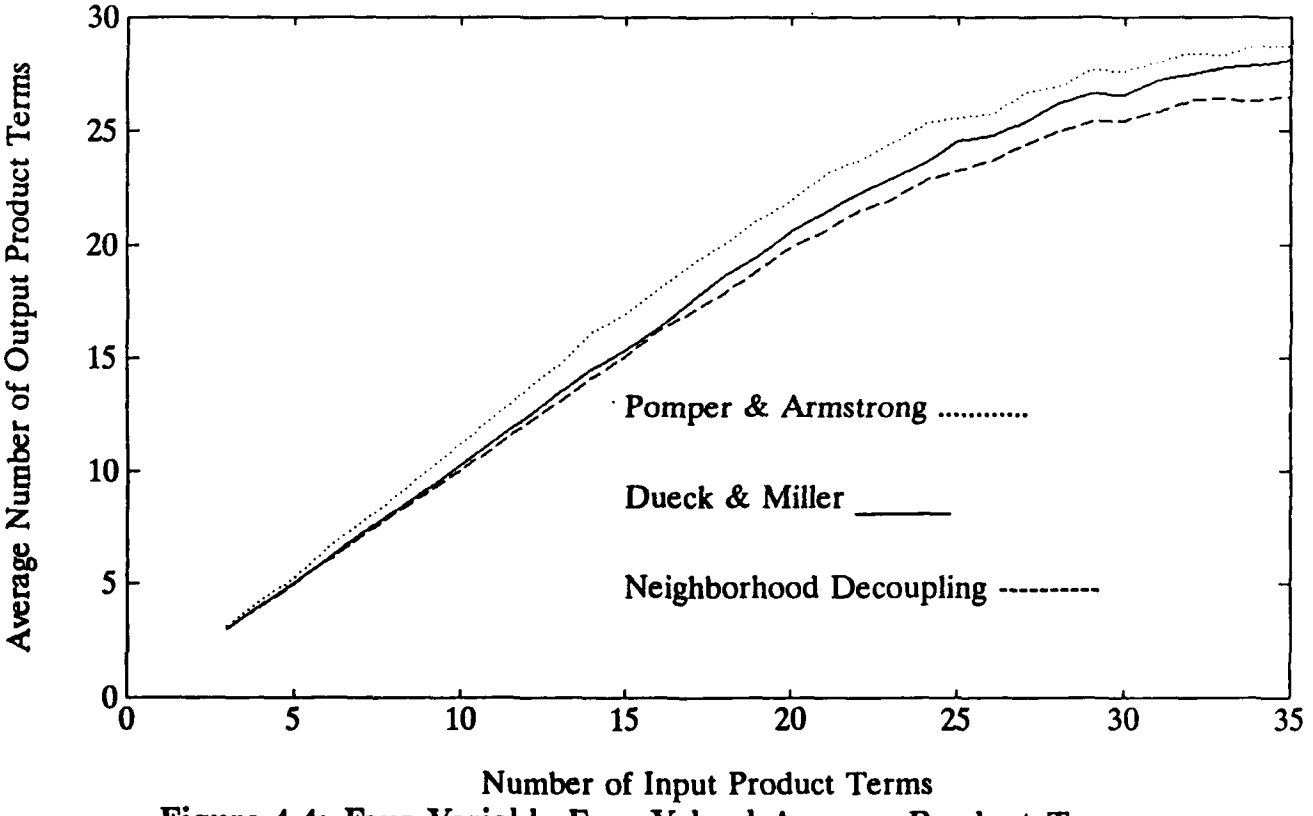

Figure 4.4: Four-Variable Four-Valued Average Product Term

TABLE 4.1: Test Results of 14000 2-Variable 4-Valued Sample Functions

Performance	Pomper and Armstrong	Dueck and Miller	Neighborhood Decoupling
best	54	212	336
better	479	2005	2308
equal	11002	11002	11002
total	11535	13219	13646
ratio	0.8239	0.9442	0.9747

2. PERFORMANCE RATIOS

We consider another measure, the performance ratio, to demonstrate the performance of each algorithm. First, for each algorithm, we count the number of minimized functions that specify which algorithm is the "best" of the three, i.e., the number of instances that a specific algorithm uses the minimum number of implicants (output product terms). If two algorithms use an equal number of implicants and less than the other one, we determine that they are "better" than the third one. When all three algorithms use an equal number of implicants to minimize a function, we determine that they are "equal". The performance ratio is defined as

$$\eta = (N_{best} + N_{better} + N_{equal}) \div N_{total}, \tag{4.1}$$

where N_{best}, N_{better}, and N_{equal} are the number of instances that specific algorithm performs "best", "better", and "equal" respectively.

The total number of functions tested, N_{total} in our case is 14,000, 11,500, 21,500, or 16,500. Tables 4.1, 4.2, 4.3, and 4.4 show the performance ratios for each setting. For example, in Table 4.1, with 14,000 functions tested, we counted the cases which Neighborhood Decoupling algorithm performs no worse than the others as 13,646. That is, $\eta_{NI} = 13646/14000 = 0.9747$. Tables 4.1, 4.2, 4.3, and 4.4 show that the performance is degraded when n or r is increased. However, the neighborhood decoupling algorithm obviously outperforms the others.

TABLE 4.2: Test Results of 11500 2-Variable 5-Valued Sample Functions

Performance	Pomper and Armstrong	Dueck and Miller	Neighborhood Decoupling
best	114	356	1248
better	762	1993	2509
equal	7150	7150	7150
total	8026	9499	10907
ratio	0.6979	0.8260	0.9484

TABLE 4.3: Test Results of 21500 3-Variable 4-Valued Sample Functions

Performance	Pomper and Armstrong	Dueck and Miller	Neighborhood Decoupling
best	948	1595	4578
better	2791	4307	5918
equal	7871	7871	7871
total	11610	13773	18367
ratio	0.5400	0.6406	0.8543

TABLE 4.4: Test Results of 16500 4-Variable 4-Valued Sample Functions

Performance	Pomper and Armstrong	Dueck and Miller	Neighborhood Decoupling
best	1604	3077	4952
better	1420	3106	3440
equal	2884	2884	2884
total	5908	9067	11276
ratio	0.3581	0.5495	0.6834

B. TIMING COMPARISON

HAMLET CAD (Computer Aided Design) tool has an option that can measure program's computation time. In this section, the algorithm timing comparison counts the average computation time for each input product term minimization. These measures are collected on the ISI workstations at NPS. The theoretical worst case computional complexity of each algorithm is $O(r^{2n})$ (see Chapter II). The decision rules for selecting minterm α and implicant I_α in each algorithm are different. In general, an algorithm which needs more or complex decision rules in selecting α and I_α should take longer computation time. There may be a trade-off between performance and computing time. In each computation cycle (select a minterm and select an implicant), each algorithm generates an output product term. The ND algorithm uses more complex decision rules than Dueck and Miller's. It may appear that ND algorithm should use more computation times. However, this is not true, since ND algorithm in general has less computation cycle than other algorithms so that it stop computation earlier than other algorithms. From the analysis of testing case results (see Appendix C), the average computation time of the ND algorithm is no worse than Dueck and Miller's for more than 4 second, sometimes even better.

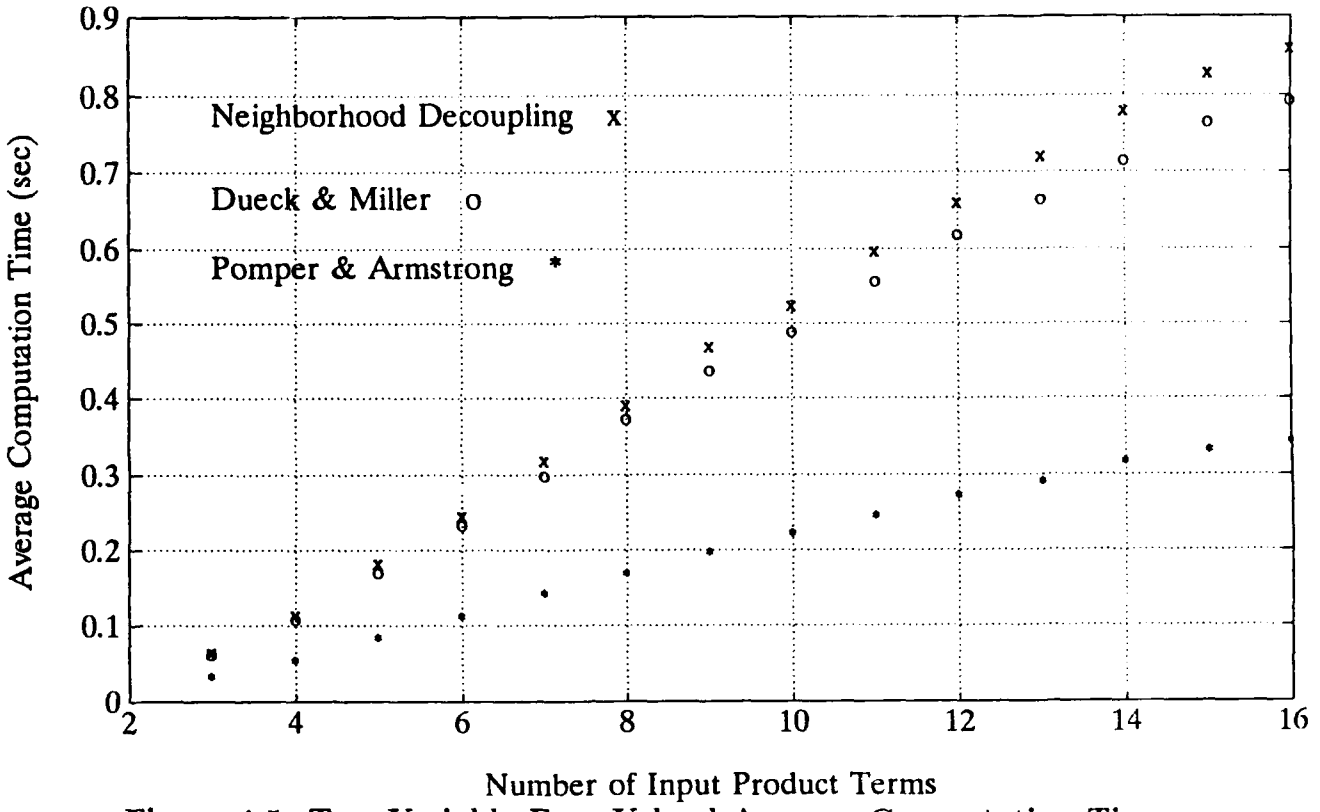

Figure 4.5: Two-Variable Four-Valued Average Computation Time

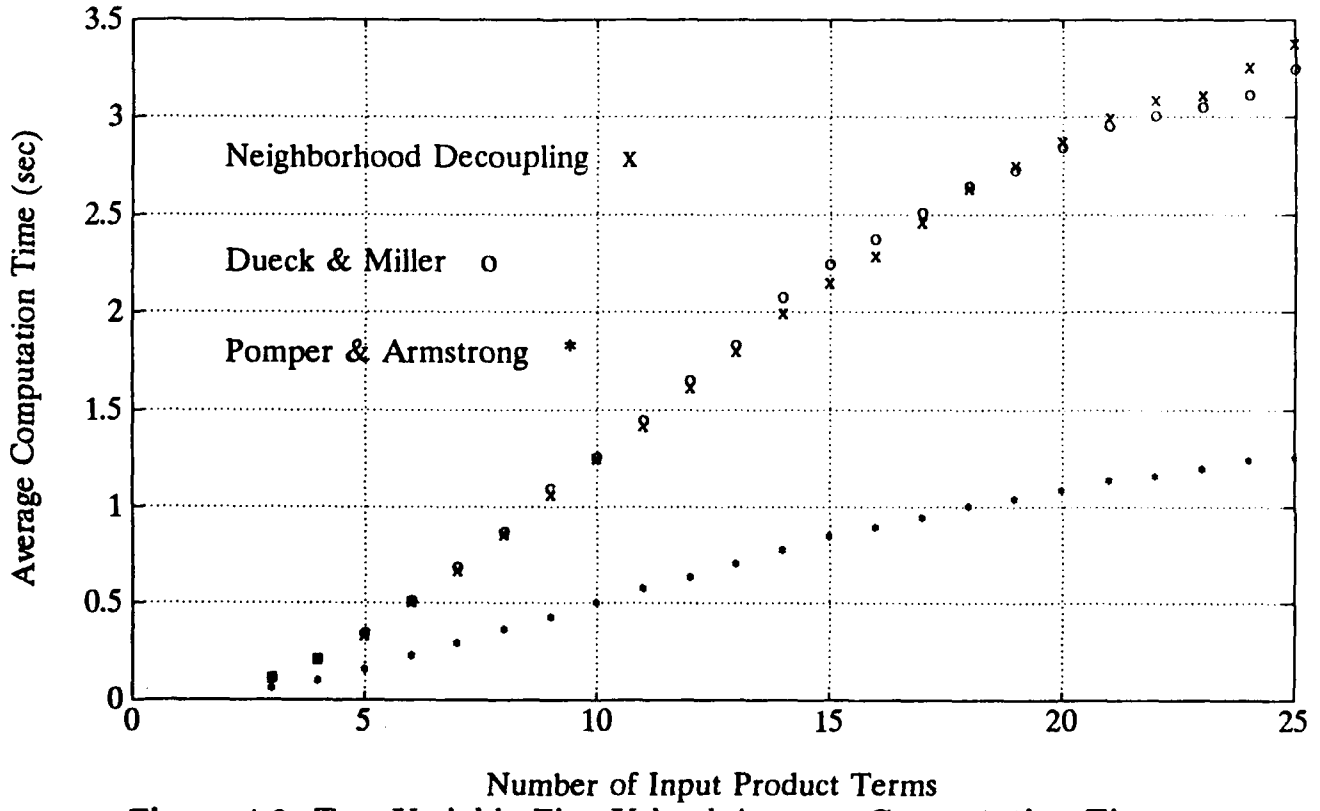

Figure 4.6: Two-Variable Five-Valued Average Computation Time

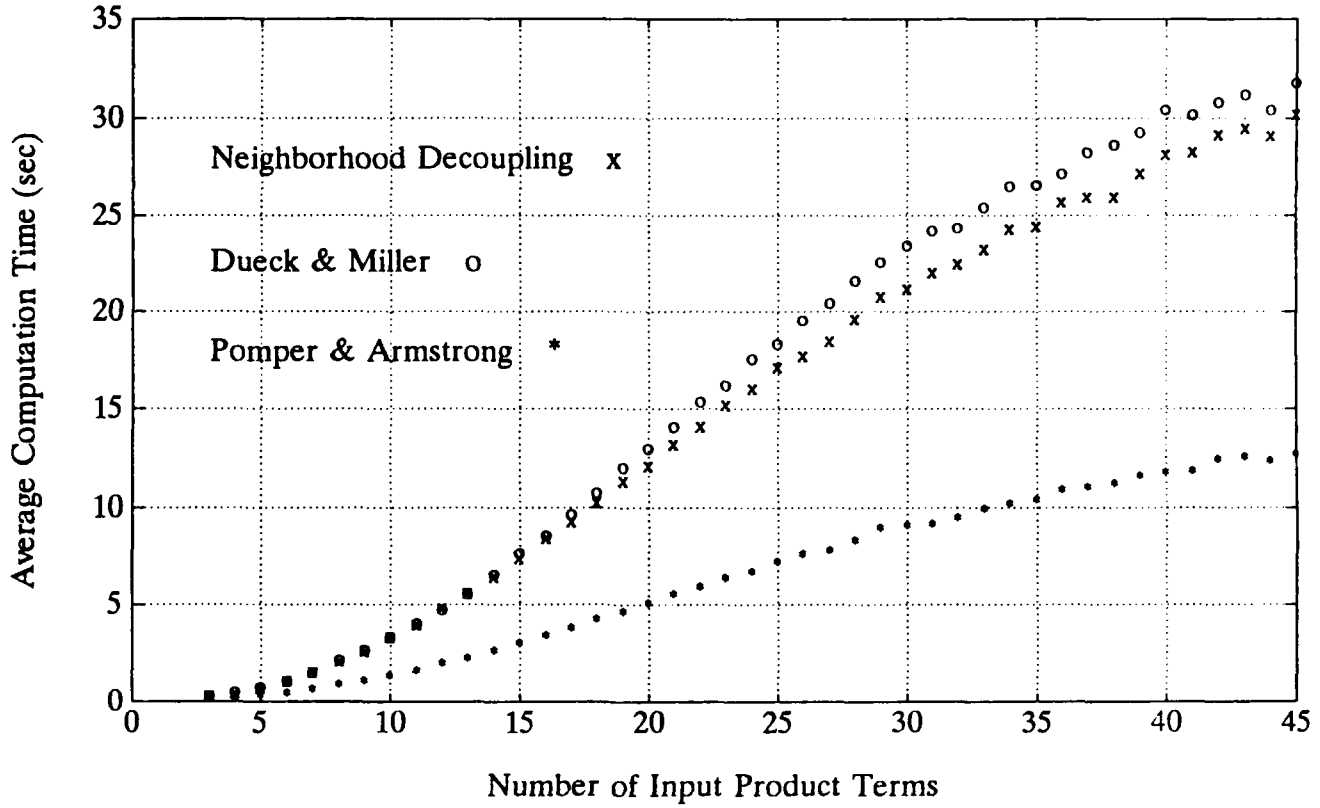

Figure 4.7: Three-Variable Four-Valued Average Computation Time

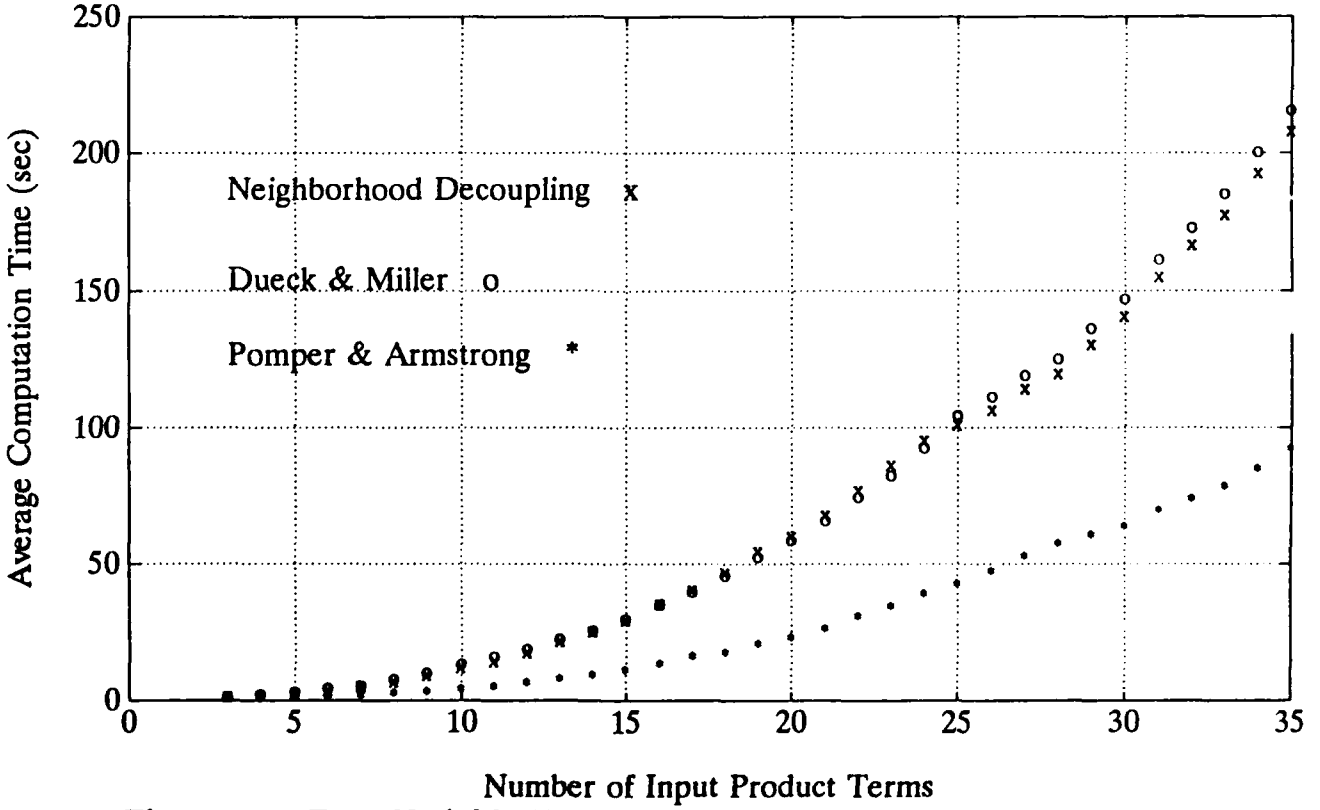

Figure 4.8: Four-Variable Four-Valued Average Computation Time

V. DISCUSSIONS AND CONCLUSIONS

A. DISCUSSIONS

Notice that in the comparisons we did not include the Bessilich's algorithm and the absolute minimum solutions. We now justify the reasons. Besslich's algorithm was not included in HAMLET because it requires a truth table to operate efficiently. We did not have room to store large truth tables. Also Besslich's algorithm is likely to require much time [Ref. 10]. In addition, we know that the Dueck and Miller is a satisfactory [Ref. 9, 10] heuristic which is reasonable approximation to the absolute minimum solutions for two-variable four-valued functions.

The performance results show that the ND algorithm outperforms the other two algorithms. Although Pomper and Armstrong's algorithm is faster than the other two algorithms, about one third of the computation time, the number of average output product terms and performance ratios are worse than the other two algorithm's. From the timing comparison results, the ND algorithm runs faster than Dueck and Miller's in larger variable setting conditions (more than two variable). This time efficiency is due to the decision rules employed in the ND algorithm which take advantage of the special property of truncated sum operations. In other words, the input product terms have a tendency to produce saturated minterms. In the decoupling process (algorithm N), a minterm in SAT will always qualify to combine with its neighbors to form an implicant. Although the ND algorithm conceptually is similar to Ducck and Miller's algorithm, the ND algorithm uses saturated minterms in an more effective way. For example, when we update (deduct) saturated minterm from expression, the minterm will be updated to a "*don't-care* minterm" (see Chapter III). Like binary logic minimization, a "*don't-care*

minterm" [Ref. 13] can simplify the minimization process.

Recall that in algorithm N, we compute the NRC values for a given minterm α by examining the relationships of I_α and its immediate neighbors, i.e., one step look-ahead. It is natural to believe that with more steps look-ahead we might make a better choice of the implicant and therefore provide a better solution. The exponential growth of the number of possible implicants restricts the practical use of k-lookahead for $k \neq 1$.

B. CONCLUSION

The truncated sum MVL minimization can be done by the neighborhood decoupling algorithm which selects the most isolated minterms as well as implicants. In the development of ND algorithm, we have the following observations: (1) truncated sum operations may produce saturated minterms, and (2) a saturated minterm will reduce to a *don't-care* minterm in the minimization process. The decision rules of the ND algorithm take full advantage of these observations. The ND algorithm outperforms most heuristic methods and does not lose its run time efficiency because the algorithm finds the solution and stops earlier than others. As the number of variable of the input function become larger, i.e., greater than three, the ability to minimize the function in reasonable time is an important and challenging research area.

APPENDIX A: PROGRAM LISTINGS

```
/* nd.c

     - This module implements the Neighborhood Decoupling heuristic for
       MVL minimization.

     - This program can be added to HAMLET [Ref. 10].

     - HAMLET is written by the instructor J. Yurchak in the Department
       of Computer Scinece at Naval Postgratuate School.

*/

#include "defs.h"

static int

       better_found;


/* Neighborhood Decoupling functions ---------------------------------- */


N_D()
/* ------------------------------------------------------------------
:function:
    - Perform the Neighborhood Decoupling heuristic on the input expression
:algorithm:
      Start with a working copy E_work of the original function E_orig;
      Initialize a final function E_final;
      While (there are still minterms to pick) {
             Pick a minterm X from E_work;
             Pick the best implicant I for X;
             Add I to E_final;
             Subtract I from E_work;
      }
:globals:
      E_orig
      e_flag
```

```
      m_flag
      q_flag
      G_flag
      FO_ratio
:side_effects:
      STAT
      HEUR
      E_work
      E_final[]
:called_by:
      main()
:calls:
      dealloc_expr()
      dup_expr()
      print_terms()
      print_map()
      mim()
      pick_implicant()
      subtract_implicant()
      print_source()
------------------------------------------------------------------------ */
{
int          num_impl = 0;
int          *X;
Implicant    *I;
float         ratio;

      if (E_final[ND].I != NULL)
          dealloc_expr(&E_final[ND]);

#ifdef ANALYZER
      STAT = &WY_stat;
#endif

      HEUR = ND;
      dup_expr(&E_work,&E_orig);
      E_final[HEUR].nterm = 0;
      E_final[HEUR].radix = E_orig.radix;
      E_final[HEUR].nvar = E_orig.nvar;
      E_final[HEUR].I = NULL;

#ifdef ANALYZER
      if (e_flag)
          print_terms(&E_orig);
      if (m_flag) {
```

```
            printf(" Orig map (nda):\n");
            print_map();
        }
#endif

        better_found = Sc_flag;

        resource_used(START);

        for (;;) {
             if ((X = mim(&E_work)) == NULL) {
             if (num_impl < E_orig.nterm)
             better_found = 1;
             break;
             }
             I = pick_implicant(X);
             num_impl++;
             subtract_implicant(I);

#ifdef ANALYZER
             if (i_flag)
             print_implicant(X,I);
             if (m_flag)
             print_map();
#endif
             if (Sm_flag) {
                 if (num_impl >= E_orig.nterm)
                     break;
             }
        }
        resource_used(STOP);

        if (!better_found) {
            num_impl = E_orig.nterm;
            dup_expr(&(E_final[ND]),&E_orig);
        }


#ifdef ANALYZER
        if (!q_flag && !G_flag) {
            if (!better_found)
            printf("  %-4d nda: %4d /%-4d %61d.%2.21d\n\n",
                   expr_seq,num_impl,num_impl,secs_used(),tsecs_used());
        else
            printf("  %-4d nda: %4d /%-4d %6d.%2.21d\n\n",
```

```
                expr_seq,num_impl,E_orig.nterm,
                secs_used(),tsecs_used());
        }
#endif

        dealloc_expr(&E_work);
}


static in       *mim(E)
Expression *E;
/* ---------------------------------------------------------------------
:function:
        - Compute the clustering factor and find the Most Isolated Minterm
          in the expression pointed to by E, and return its coordinates as
          a vector.
        - Local to nd.c
:globals:
        radix
        nvar
:side_effects:
        STAT
:called_by:
        N_D()
:calls:
        next_coord()
        eval()
        vcopy()
:returns:
        - A vector of integers representing the coordinate of the most
          isolated minterm, or NULL if no more minterms.
        - The value at that location is also returned as the last integer
          in the vector.
--------------------------------------------------------------------- */
{
register      i,j,k;
int           cur_val = E->radix,
              cur_CF = MAX_INT,
              X_orig[MAX_VAR+2],
              R_1 = radix - 1,
              Not_all = 0,
              All_trun = 0,
              TRUN = 2*R_1,
              last = 0,
              expanded,
```

```
                value[2],
                val1[2],
                val2[2],
                cf,
                ea,
                dea,
                term;
int             *X,*next_coord();
static int
                coord[MAX_VAR+2],
                save_coord[MAX_VAR+2];

#ifdef ANALYZER
                STAT->calls_mim++;
#endif

 for (term=0; term < E_orig.nterm; term++) {
    k = 1;
    while ((X=next_coord(coord,&(E->I[term]),k)) != NULL) {
        vcopy(value,eval(E,X));
        if (value[EVAL] && value[EVAL] < radix) {
            if (!value[HLV])
                Not_all = 1;
                if (All_trun) {
                    cf = 0;
                    dea = 0;
                    ea = 0;
                    for (j=0; j < nvar; j++)  X_orig[j] = X[j];
                /* for each variable (direction)... */
                    for (j=0; j < nvar; j++ ) {
                         expanded = 0;
                /* If not on a left hand edge, move left */
                         while (X[j] > 0) {
                                X[j]--;
                                vcopy(val1,eval(E,X));
                                if (val1[EVAL]) {
                                    expanded = 1;
                                    ea++;
                                }
                                else
                                    break;
                          }
                          X[j] = X_orig[j];
                          if (expanded) {
                              expanded = 0;
```

```
                dea++;
            }
/* if we didn't start on a right hand edge, move right */
            while (X[j] < R_1) {
                    X[j]++;
                    vcopy(val2,eval(E,X));
                    if (val2[EVAL]) {
                        expanded = 1;
                        ea++;
                    }
                    else
                        break;
             }
             X[j] = X_orig[j];
             if (expanded)
                 dea++;
    }

/* compute the clustering factor */
    cf = (dea * R_1) + ea;
    if (cf < cur_CF) {
        cur_val = value[EVAL];
        cur_CF = cf;
        for (i=0; i < nvar; i++) save_coord[i] = X[i];
    }
}
else {

    cf = 0;
    dea = 0;
    ea = 0;
    for (j=0; j < nvar; j++)  X_orig[j] = X[j];
/* for each variable (direction)... */
    for (j=0; j < nvar; j++ ) {
         expanded = 0;
/* If not on a left hand edge, move left */
         while (X[j] > 0) {
                X[j]--;
                vcopy(val1,eval(E,X));
                if (val1[EVAL] && (val1[EVAL] <= value[EVAL]
                    || value[HLV])) {
                    expanded = 1;
                    ea++;
                }
                else
```

```
                        break;
                    }
                    X[j] = X_orig[j];
                    if (expanded) {
                        expanded = 0;
                        dea++;
                    }
            /* if we didn't start on a right hand edge, move right */
                    while (X[j] < R_1) {
                        X[j]++;
                        vcopy(val2,eval(E,X));
                        if (val2[EVAL] && (val2[EVAL] <= value[EVAL]
                            || value[HLV])) {
                            expanded = 1;
                            ea++;
                        }
                        else
                            break;
                    }
                    X[j] = X_orig[j];
                    if (expanded)
                        dea++;
                }

        /* compute the clustering factor */
                cf = (dea * R_1) + ea;

                if (!(value[HLV] && cf > TRUN)) {
                    if (cf < cur_CF) {
                        cur_val = value[EVAL];
                        cur_CF = cf;
                        for (i=0; i < nvar; i++) save_coord[i] = X[i];
                    }
                }
            }
        }   /* end if */
        k = 0;
    }   /* end while */

    if (!last && (term == (E_orig.nterm - 1)) && !Not_all) {
        All_trun = 1;
        cur_CF = MAX_INT;
        term = -1;
        last = 1;
    }
```

```
  }
  if (cur_CF == MAX_INT)
      return(NULL);
      save_coord[nvar+1] = cur_CF;
      save_coord[nvar] = cur_val;

      return(save_coord);
}


static int      valid_implicant(I)
Implicant       *I;
/* ---------------------------------------------------------------------
:function:
         - Decide upon the validity of implicant I
         - Local to nd.c
:globals:
         E_work
         E_orig
:side_effects:
         STAT
:called_by:
         pick_implicant()
:calls:
         next_coord()
         eval()
         vcopy()
:returns:
         1 if a valid implicant
         0 if not
--------------------------------------------------------------------- */
{
int
       *X,
       init = 1,
       R_1 = radix - 1,
       value = I->coeff,
       Vo[2],Vw[2];
static int
       coord[MAX_VAR+2];

#ifdef ANALYZER
       STAT->calls_valid_implicant++;
#endif
```

```
    while ((X = next_coord(coord,I,init)) != NULL) {
            init = 0;
            vcopy(Vw,eval(&E_work,X));
            vcopy(Vo,eval(&E_orig,X));
            if (((Vw[EVAL] < value) && !Vw[HLV]) && (Vo[EVAL] < R_1))
            return(0);
    }
    return(1);
}


static int      compute_nrc(I)
Implicant       *I;
/* ---------------------------------------------------------------------
:function:
        - Compute the NRC for the given implicant
        - Local to nd.c
:globals:
        radix
        nvar
:side_effects:
        STAT
:called_by:
        pick_implicant()
:calls:
        next_coord()
        eval()
        vcopy()
:returns:
        - an integer NRC
--------------------------------------------------------------------- */
{
int      *X;
int      I_value = I->coeff;
register i;
int      value[2],
         R_1 = radix - 1,
         neighbor_value[2],
         good,
         bad,
         diff,
         equal,
         neig_boun,
         first,
         nrc = 0,
```

```
        init = 1;
static int
        coord[MAX_VAR+2];

#ifdef ANALYZER
        STAT->calls_compute_nrc++;
#endif

          /* for each coordinate in the implicant ... */
    while ((X = next_coord(coord,I,init)) != NULL) {
          init = 0;
          equal = 0;
          vcopy(value,eval(&E_work,X));
          if (value[EVAL] == radix)
              continue;
              diff = value[EVAL] - I_value;
              first = 1;
          /* for each direction ... */
              for (i=0; i < nvar; i++) {
                   good = 0;
                   bad = 0;
                   if ((diff <= 0) && first) {
                        good =  2;
                        first = 0;
                   }
          /* if there is a left neighbor, examine it */
                   if (X[i] != 0 && X[i] == I->B[i].lower) {
                       X[i]--;
                       vcopy(neighbor_value,eval(&E_work,X));
                       neig_boun = neighbor_value[EVAL] - value[EVAL];
                       X[i]++;
                       if (neighbor_value[EVAL] != 0) {
                           if (!neighbor_value[HLV] || !value[HLV]) {
                               if (neighbor_value[EVAL] < diff) {
                                   if (neighbor_value[HLV])
                                       good += 1;
                                   else
                                       bad += 2;
                               }
                               if (neighbor_value[EVAL] > diff) {
                                   if (!neig_boun)
                                       bad += 2;
                                   if (neighbor_value[HLV] && neig_boun < 0)
                                       bad += 2;
                                   if (diff > 0 && neig_boun) {
```

```
                                if (value[HLV])
                                    good += 1;
                                else
                                    bad += 2;
                            }
                        }
                        else {
                            if (neighbor_value[HLV] || value[HLV])
                                good += 1;
                            else
                                good += 2;
                        }
                    }
                } /* end if */
            }

        /* if there is a right neighbor, examine it */
            if (X[i] != R_1 && X[i] == I->B[i].upper) {
                X[i]++;
                vcopy(neighbor_value,eval(&E_work,X));
                neig_boun = neighbor_value[EVAL] - value[EVAL];
                X[i]--;
                if (neighbor_value[EVAL] != 0) {
                    if (!neighbor_value[HLV] || !value[HLV]) {
                        if (neighbor_value[EVAL] < diff) {
                            if (neighbor_value[HLV])
                                good += 1;
                            else
                                bad += 2;
                        }
                        if (neighbor_value[EVAL] > diff) {
                            if (!neig_boun)
                                bad += 2;
                            if (neighbor_value[HLV] && neig_boun < 0)
                                    bad += 2;
                            if (diff > 0 && neig_boun) {
                                if (value[HLV])
                                    good += 1;
                                else
                                    bad += 2;
                            }
                        }
                        else {
                            if (neighbor_value[HLV] || value[HLV])
                                good += 1;
```

```
                                    else
                                        good += 2;
                                }
                            }
                        }
                    }

                /* update the nrc */
                        nrc = (nrc - good) + bad;
                  } /* end for */
    } /* end while */
    return(nrc);
}


static Implicant        *pick_implicant(X)
int                     *X;
/* ---------------------------------------------------------------------
:function:
          - Pick the best implicant for minterm X
:globals:
         radix
:side_effects:
         STAT
:called_by:
         N_D()
:calls:
         init_implicant()
         gen_bounds()
         next_implicant()
         eval()
         vcopy()
         compue_nrc()
         copy_implicant()
         valid_implicant()
:returns:
         - A pointer to a term representing the best implicant.
--------------------------------------------------------------------- */
{
int      cur_nrc = MAX_INT,
         nrc = 0,
         I_value,
         i,
         init = 1,
         first = 1;
```

```
Implicant
        *I;
        static int
        coord[MAX_VAR+2];
static Bound
        I_bound[MAX_VAR+2];
static Implicant
        I_best;
Bound   *B;
int     V[2],
        value[2];

#ifdef ANALYZER
        STAT->calls_pick_implicant++;
#endif

   I_best.B = I_bound;
   init_implicant(X);
   B = gen_bounds(X);
   vcopy(V,eval(&E_orig,X));
   while ((I = next_implicant(B)) != NULL) {
          if (V[HLV]) {
              for (I->coeff=X[nvar]; I->coeff < radix; (I->coeff)++) {
                   if (valid_implicant(I)) {
                       nrc = compute_nrc(I);
                       if (first)
                           nrc = 2;
                       else
                           nrc += 2;
                       if (nrc <= cur_nrc) {
                           cur_nrc = nrc;
                           I->nrc = nrc;
                           copy_implicant(&I_best,I);
                       }
                   }
              }
              first = 0;
          }
          else {
              I->coeff = X[nvar];
              if (valid_implicant(I)) {
                  nrc = compute_nrc(I);
                  if (first) {
                      first = 0;
                      if (nrc < 0 )
```

```
                        nrc = 1;
                    else
                        nrc += 2;
                }
                else
                    nrc += 2;
                    if (nrc <= cur_nrc) {
                        cur_nrc = nrc;
                        I->nrc = nrc;
                        copy_implicant(&I_best,I);
                    }
            }
        }
    }
    return(&I_best);
}
```

APPENDIX B: AVERAGE NUMBER OF OUTPUT PRODUCT TERMS

TABLE B.1: Two-Variable Four-Valued Average Output Product Terms

Number of Input Terms	Pomper and Armstrong	Dueck and Miller	Neighborhood Decoupling
3	2.838	2.716	2.709
4	3.483	3.290	3.277
5	3.916	3.710	3.675
6	4.178	3.959	3.934
7	4.203	4.023	3.977
8	4.201	4.009	3.950
9	4.072	3.920	3.872
10	3.913	3.763	3.699
11	3.717	3.583	3.534
12	3.573	3.438	3.398
13	3.355	3.233	3.207
14	3.178	3.079	3.042
15	2.991	2.899	2.877
16	2.759	2.696	2.658

TABLE B.2: Two-Variable Five-Valued Average Output Product Terms

Number of Input Terms	Pomper and Armstrong	Dueck and Miller	Neighborhood Decoupling
3	3.122	2.858	2.832
4	4.044	3.718	3.684
5	4.660	4.296	4.236
6	5.222	4.868	4.760
7	5.540	5.138	5.040
8	5.740	5.428	5.258
9	5.832	5.548	5.360
10	5.948	5.566	5.420
11	5.814	5.530	5.372
12	5.762	5.524	5.306
13	5.486	5.246	5.062
14	5.364	5.198	4.894
15	5.014	4.858	4.622
16	4.780	4.668	4.376
17	4.484	4.360	4.172
18	4.250	4.184	4.016
19	4.032	3.942	3.762
20	3.822	3.766	3.600
21	3.730	3.676	3.504
22	3.564	3.482	3.362
23	3.224	3.152	3.032
24	3.044	2.998	2.902
25	2.982	2.944	2.840

TABLE B.3: Three-Variable Four-Valued Average Output Product Terms

Number of Input Terms	Pomper and Armstrong	Dueck and Miller	Neighborhood Decoupling
3	3.026	2.940	2.928
4	4.090	3.892	3.862
5	5.162	4.834	4.840
6	6.168	5.756	5.718
7	7.186	6.578	6.492
8	7.916	7.372	7.238
9	8.466	7.852	7.762
10	9.02	8.570	8.324
11	9.646	9.068	8.814
12	10.056	9.578	9.266
13	10.426	9.918	9.608
14	10.588	10.126	9.772
15	10.730	10.280	9.958
16	10.668	10.330	9.938
17	10.676	10.450	10.012
18	10.866	10.634	10.124
19	10.628	10.482	9.870
20	10.586	10.358	9.796
21	10.508	10.390	9.804
22	10.280	10.240	9.586
23	10.164	9.990	9.364
24	9.726	9.628	8.990
25	9.448	9.426	8.806
26	9.142	9.098	8.492
27	8.896	8.948	8.244
28	8.706	8.756	8.098
29	8.820	8.834	8.202
30	8.648	8.746	7.976
31	8.592	8.688	7.968
32	8.104	8.154	7.544
33	8.022	8.070	7.444
34	7.646	7.766	7.188
35	7.160	7.174	6.684
36	7.076	7.186	6.676
37	6.888	6.994	6.388
38	6.632	6.642	6.132
39	6.492	6.528	6.036
40	6.406	6.350	5.926
41	6.128	6.214	5.750
42	5.878	5.864	5.502
43	5.570	5.664	5.224
44	5.270	5.286	4.940
45	5.302	5.340	4.960

TABLE B.4: Four-Variable Four-Valued Average Output Product Terms

Number of Input Terms	Pomper and Armstrong	Dueck and Miller	Neighborhood Decoupling
3	3.136	3.018	3.002
4	4.252	4.026	3.974
5	5.248	5.028	4.978
6	6.546	6.070	6.018
7	7.690	7.168	7.054
8	8.786	8.178	8.066
9	9.988	9.172	8.970
10	11.172	10.246	10.014
11	12.354	11.330	10.960
12	13.600	12.374	12.074
13	14.676	13.502	13.066
14	16.096	14.512	14.144
15	16.954	15.366	15.056
16	18.028	16.348	16.230
17	19.144	17.570	17.036
18	20.050	18.648	17.866
19	21.072	19.460	18.862
20	21.930	20.590	19.850
21	23.074	21.396	20.576
22	23.682	22.258	21.456
23	24.456	22.910	21.954
24	25.258	23.590	22.770
25	25.528	24.552	23.220
26	25.740	24.788	23.654
27	26.656	25.412	24.374
28	26.940	26.190	24.950
29	27.786	26.668	25.410
30	27.672	26.612	25.454
31	28.032	27.246	25.854
32	28.458	27.550	26.318
33	28.368	27.840	26.454
34	28.768	27.934	26.344
35	28.752	28.170	26.498

APPENDIX C: AVERAGE COMPUTATION TIME FOR MVL MINIMIZATION

TABLE C.1: Two-Variable Four-Valued Average Computation Time

Number of Input Terms	Pomper and Armstrong	Dueck and Miller	Neighborhood Decoupling
3	0.03309	0.06156	0.06336
4	0.05517	0.10724	0.11221
5	0.08384	0.16931	0.17947
6	0.11167	0.23128	0.24318
7	0.14140	0.29612	0.31576
8	0.1[illegible]0	0.37093	0.38907
9	0.19713	0.43512	0.46649
10	0.22143	0.48732	0.52141
11	0.24440	0.55478	0.59490
12	0.27069	0.61506	0.65662
13	0.28888	0.66258	0.71877
14	0.31570	0.71373	0.77805
15	0.33116	0.76289	0.82645
16	0.34269	0.79065	0.85818

TABLE C.2: Two-Variable Five-Valued Average Computation Time

Number of Input Terms	Pomper and Armstrong	Dueck and Miller	Neighborhood Decoupling
3	0.05604	0.10954	0.11110
4	0.09554	0.20432	0.20372
5	0.15442	0.33730	0.33034
6	0.22760	0.50900	0.50168
7	0.29088	0.67988	0.66278
8	0.36180	0.86232	0.84642
9	0.42612	1.08536	1.05530
10	0.50258	1.24780	1.23926
11	0.57016	1.44212	1.41004
12	0.62940	1.64508	1.60892
13	0.70374	1.83080	1.79734
14	0.76936	2.07240	1.98956
15	0.84158	2.23954	2.14072
16	0.88608	2.36738	2.27968
17	0.93786	2.50298	2.45116
18	0.99650	2.63846	2.62578
19	1.03754	2.72054	2.74588
20	1.07786	2.84114	2.86624
21	1.13426	2.94840	2.98710
22	1.15298	2.99784	3.08048
23	1.18984	3.04716	3.10488
24	1.23294	3.10796	3.25222
25	1.24788	3.24612	3.37128

TABLE C.3: Three-Variable Four-Valued Average Computation Time

Number of Input Terms	Pomper and Armstrong	Dueck and Miller	Neighborhood Decoupling
3	0.11096	0.25190	0.23496
4	0.17684	0.41760	0.39676
5	0.29150	0.67308	0.66464
6	0.43990	1.04038	1.02516
7	0.65294	1.49640	1.47802
8	0.86260	2.09008	2.07168
9	1.07620	2.61164	2.59776
10	1.36058	3.29064	3.24702
11	1.63312	3.97860	3.93274
12	1.97194	4.71946	4.73014
13	2.24472	5.53536	5.50418
14	2.61166	6.50490	6.37408
15	3.00044	7.56094	7.31712
16	3.40472	8.50130	8.33298
17	3.80686	9.56092	9.21312
18	4.22300	10.6681	10.1814
19	4.62690	11.9961	11.2720
20	5.10534	12.9534	12.0204
21	5.51390	14.0393	3.1130
22	5.91520	15.2872	4.0559
23	6.35928	16.1587	5.1375
24	6.69882	17.5192	15.9774
25	7.17424	18.2629	17.0413
26	7.60032	19.5004	17.6663
27	7.77850	20.4014	18.4309
28	8.32284	21.5390	19.5850
29	8.95526	22.5366	20.7064
30	9.04034	23.3790	21.0932
31	9.11648	24.1279	21.9608
32	9.48198	24.2701	22.3971
33	9.88210	25.3621	23.1439
34	10.1734	26.4659	24.2141
35	10.3463	26.5370	24.3088
36	10.8508	27.1308	25.6463
37	11.0079	28.1984	25.9083
38	11.2138	28.6137	25.8980
39	11.5521	29.2124	27.1013
40	11.7711	30.3718	28.0957
41	11.8719	30.1055	28.1734
42	12.4192	30.7879	29.1146
43	12.5431	31.1774	29.4338
44	12.3511	30.4067	29.0054
45	12.6780	31.7510	30.1296

TABLE C.4: Four-Variable Four-Valued Average Computation Time

Number of Input Terms	Pomper and Armstrong	Dueck and Miller	Neighborhood Decoupling
3	0.35778	1.01774	0.79966
4	0.57068	1.64304	1.33264
5	0.89762	2.74510	2.17928
6	1.26432	3.97440	3.24550
7	1.69976	5.14632	4.58174
8	2.52112	7.31790	6.54000
9	3.25296	9.42768	8.51408
10	4.23814	12.6106	11.3127
11	5.21906	15.5529	13.8771
12	6.31822	18.5009	17.0987
13	7.86472	22.1662	20.9875
14	9.35484	25.1544	24.6584
15	10.9688	29.2464	29.0192
16	13.0702	34.6918	34.6770
17	15.9150	39.5156	40.0190
18	17.5832	45.6882	46.5026
19	20.8329	52.2132	54.0485
20	22.9053	58.4218	59.8628
21	26.1380	65.7590	67.4621
22	30.5844	74.2779	76.7863
23	34.5081	82.4673	86.0243
24	39.0754	92.3658	94.9414
25	42.8344	104.355	100.501
26	47.3552	111.323	106.223
27	53.0040	118.979	113.879
28	57.5171	125.230	119.541
29	60.7629	136.164	129.975
30	63.8871	146.791	140.376
31	70.0774	160.949	154.421
32	73.9980	172.364	165.895
33	78.4594	185.003	177.013
34	84.8676	200.174	191.966
35	92.5689	215.201	207.516

REFERENCES

1. K. C. Smith, "The prospect for multivalued logic: a technology and applications view," *IEEE Trans. Computers*, pp. 619-632, Dec. 1981

2. S. L. Hurst, "Multiple-valued logic - its status and its future," *IEEE Trans. Computers*, Vol c-33, pp. 1160-1179, Dec. 1984

3. H. G. Kerkhoff, "Theory and design of multiple-valued logic CCD's," in Computer Science and Multiple-Valued Logic (ed. D. C. Rine), North Holland, New York, pp. 502-537, 1984

4. J. Butler and H. G. Kerkhoff, "Multiple-valued CCD circuits," *IEEE Computer*, pp. 58-69, March, 1988

5. G. Pomper and J. A. Armstrong, "Representation of multivalued functions using the direct cover method," *IEEE Trans. Comp.*, pp. 674-679, Sept. 1981

6. P. W. Besslich, "Heuristic minimization of MVL functions: a direct cover approach," *IEEE Trans. Comp.*, Vol C-35, pp. 134-144, Feb. 1986.

7. G. W. Dueck and D. M. Miller, "A direct cover MVL minimization using the truncated sum," *Proc. of 17th Intl. Symp. on MVL*, pp. 221-226, 1987

8. G. W. Dueck, "Algorithms for the minimization of binary and multiple-valued logic functions," Ph. D. Dissertation, Department of Computer Science, University of Manitoba, Winnipeg, MB, 1988

9. P. Tirumalai and J. T. Butler, "Analysis of minimization algorithms for multiple-valued PLA," *Proc. of 18th Intl. Symp. on MVL*, pp. 226-236, 1988

10. J. Yurchak and J. T. Butler, "HAMLET - An expression compiler/optimizer for the implementation of heuristics to minimize multiple-valued programmable logic arrays," Draft, submitted to the *Intl. Symp. on MVL*, 1990

11. E. A. Bender, J. Butler, and H. G. Kerkhoff, "Comparing the SUM with the MAX for use in four-valued PLA'S," *Intl. Symp. on MVL*, pp. 30-35, May 1985

12. A. V. Aho, J. E. Hopcroft, and J. D. Ullman, *The Design and Analysis of Computer Algorithms*, Reading, Mass., Addison-Wesley, 1974

13. E. J. McCluskey, ***Logic Design Principles***, Englewood Cliffs, NJ, Prentice-Hall, 1986

INITIAL DISTRIBUTION LIST

		No. of Copies
1.	Defense Technical Information Center Cameron Station Alexandria, Virginia 22304-6145	2
2.	Library, Code 0142 Naval Postgraduate School Monterey, California 93943-5002	2
3.	C.C.I.T. Library P.O.Box, 90047, Ta-Hsi Tao-Yuan, Taiwan, R.O.C.	2
4.	Chairman, Code 62 Department of Electrical and Computer Engineering Naval Postgraduate School Monterey, California 93943-5000	1
5.	Chairman Department of Electrical Engineering C.C.I.T. P.O.Box 90047-43 Ta-Hsi Tao-Yuan, Taiwan, R.O.C.	2
6.	Naval Postgraduate School Attn: Professor Chyan Yang, Code 62Ya Monterey, California 93943-5000	6
7.	Naval Postgraduate School Attn: Professor Jon T. Butler, Code 62Bu Monterey, California 93943-5004	4
8.	Naval Postgraduate School Attn: Instructor John M. Yurchak, Code 52Yu Monterey, California 93943-5000	1

No. of Copies

9. Lt. Comdr. Hsian-Han Meng — 2
Department of Weapon Engineering
C.C.I.T. P.O.Box 90047-25 Ta-Hsi
Tao-Yuan, Taiwan, R.O.C.

10. CPT Yao-Ming Wang — 2
F4 3-3 ALY. 11 LN. 587 Weng-Lin RD. Shih-Lin
Taipei, Taiwan, R.O.C.